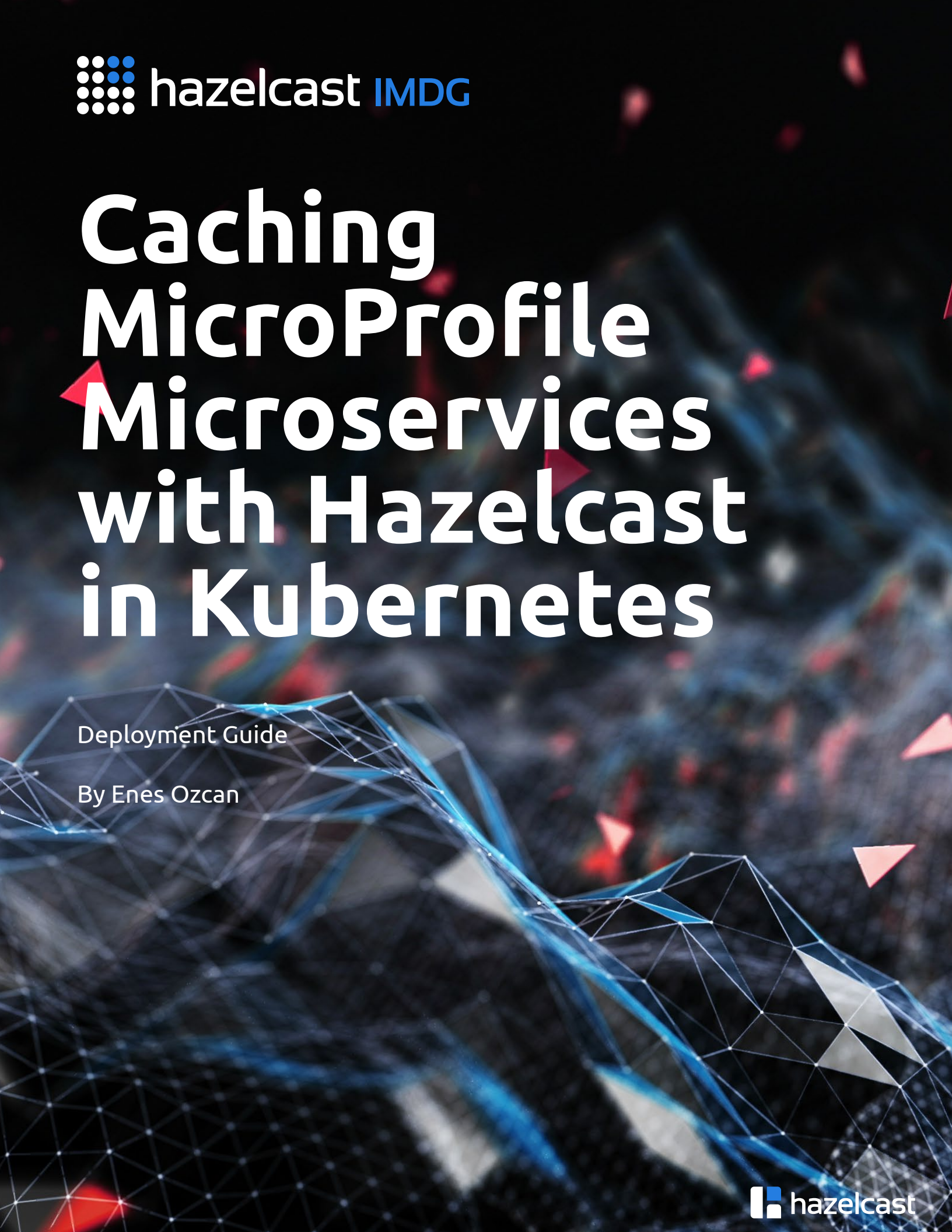
hazelcast IMDG
Caching MicroProfile Microservices with Hazelcast in Kubernetes
Deployment Guide
By Enes Ozcan
hazelcast

Table of Contents

Caching MicroProfile Microservices with Hazelcast in Kubernetes

You can see the whole project here and start building your app in the final directory.

What You'll Learn

In this guide, you will learn how to use Hazelcast distributed caching with MicroProfile and deploy to a local Kubernetes cluster. You will then create a Kubernetes Service which load balances between containers and verify that you can share data between microservices.

The microservice you will deploy is called `hazelcast-microprofile`. The `hazelcast-microprofile` microservice simply helps you write ("put") data and read it back. As the Kubernetes Service sends the request to a different pod each time you initiate the request, the data will be served by the shared hazelcast cluster between `hazelcast-microprofile` pods.

You will use a local single-node Kubernetes cluster. However, you can deploy this application on any Kubernetes distribution.

What Is Hazelcast?

Hazelcast is an open source in-memory data grid (IMDG). It provides elastically scalable, distributed in-memory computing, widely recognized as the fastest and most scalable approach to application performance.

Hazelcast is designed to scale up to hundreds and thousands of members. Simply add new members and they will automatically discover the cluster and will linearly increase both memory and processing capacity.

Why MicroProfile?

The MicroProfile is a baseline platform definition that optimizes Enterprise Java for a microservices architecture and delivers application portability across multiple MicroProfile runtimes. To learn more about MicroProfile, visit this website.

Prerequisites

Before you begin, have the following tools installed:

- You will need Apache Maven to build and run the project.
- You will also need containerization software for building containers. Kubernetes supports a variety of container types. We will use `Docker` in this guide. For installation instructions, refer to the official Docker documentation.

For Windows and Mac

- Use Docker Desktop, where a local Kubernetes environment is pre-installed and enabled. Download it from the official website.

Linux

- Use `Minikube` as a single-node Kubernetes cluster that runs locally in a virtual machine. For Minikube installation instructions, please visit this page.

Getting Started

The fastest way to work through this guide is to clone the Git repository and use the projects that are provided inside:

```
$ git clone https://github.com/enozcan/guide-kubernetes-caching-hazelcast-microprofile
$ cd guide-kubernetes-caching-hazelcast-microprofile
```

The `initial` directory contains the starting project that you will build upon.

The `final` directory contains the finished project you will build.

Running a MicroProfile Application

The application in the initial directory is a basic MicroProfile app having a few endpoints. We are going to use only **/application/map/put** and **/application/map/get** endpoints through this guide.

Build and run the app using Maven in the `initial` directory:

```
$ > mvn install liberty:run-server
```

When the log **"The GettingStartedServer server is ready to run a smarter planet."** is seen, the app is ready and running on localhost:9080. You can test by following requests:

```
$ > curl "http://localhost:9080/application/map/put?key=key_1&value=hazelcast"

$ > curl "http://localhost:9080/application/map/get?key=key_1"
```

The first one will not return a response. The second one will return the value belonging to the key given as the parameter (`key_1` and `hazelcast` in this request) and the responding pod (currently `null`).

This part was the introduction of the application. You can stop your application by **CTRL + C**.

Dockerizing the App

To create the `Docker image` of the application, use the `docker-image` profile in the `pom.xml`. This profile will build the Docker image using the Dockerfile under the `initial` directory.

Build the app under the `initial` directory using profile:

```
$ > mvn clean package -P docker-image
```

Now, the image must be seen among the Docker images:

```
$ > docker images

REPOSITORY                          TAG           IMAGE ID         CREATED           SIZE
openliberty-hazelcast-microprofile 1.0-SNAPSHOT  5f65a62b0aa0     19 seconds ago    122MB
```

Running the App in a Container

Now that the Docker image is ready, check if the image runs properly:

```
$ > docker run -p 9080:9080 openliberty-hazelcast-microprofile:1.0-SNAPSHOT
```

Test the app on the port 9080:

```
$ > curl "http://localhost:9080/application/map/put"?key=key_1&value=hazelcast"
$ > curl "http://localhost:9080/application/map/get?key=key_1"
```

If you see the same responses as the ones you get when the app is run without container, that means it's all OK with the image.

To stop the container, get the container ID first:

```
$ > docker ps
```

Then find the application's container ID and stop the container:

```
$ > docker stop [CONTAINER-ID]
```

Starting and Preparing Your Cluster for Deployment

Now that you have a proper Docker image, deploy the app to Kubernetes pods. Start your Kubernetes cluster first.

Windows | Mac

Start your Docker Desktop environment. Make sure "Docker Desktop is running" and "Kubernetes is running" status are updated.

Linux

Run the following from command line:

```
$ > minikube start
```

Validate Kubernetes Environment

Next, validate that you have a healthy Kubernetes environment by running the following command from the command line.

```
$ > kubectl get nodes
```

This command should return a `Ready` status for the master node.

Windows | Mac

You do not need to do any other step.

Linux

Run the following command to configure the Docker CLI to use Minikube's Docker daemon.

After you run this command, you will be able to interact with Minikube's Docker daemon and build new images directly to it from your host machine:

```
$ > eval $(minikube docker-env)
```

After you're sure that a master node is ready, create kubernetes.yaml under `initial` directory with the same content in the `final/kubernetes.yaml` file.

This file defines two Kubernetes resources: one StatefulSet and one service. StatefulSet is preferred solution for Hazelcast because it enables controlled scale out/in of your microservices for easy data distribution. To learn more about StatefulSet, you can visit Kubernetes documentation.

By default, we create 2 replicas of the `hazelcast-microprofile` microservice behind `hazelcast-microprofile-service` which forwards requests to one of the pods available in the Kubernetes cluster.

`MY_POD_NAME` is an environment variable made available to the pods so that each microservice knows which pod they are in. This is going to be used in this guide in order to show which pod is responding to the HTTP request. It's fetched and used in the `MapResource.java` file.

Run the following command to deploy the resources as defined in kubernetes.yaml:

```
$ > kubectl apply -f kubernetes.yaml
```

Run the following command to check the status of your pods:

```
$ > kubectl get pods
```

You'll see an output similar to the following if all the pods are healthy and running:

```
NAME                                    READY         STATUS        RESTARTS      AGE
hazelcast-microprofile-statefulset-0    1/1           Running       0             7s
hazelcast-microprofile-statefulset-1    1/1           Running       0             5s
```

Even if the status of the nodes are `Running`, they might be not started yet. Check the pod logs to be sure they're ready:

```
$ > kubectl logs hazelcast-microprofile-statefulset-0

$ > kubectl logs hazelcast-microprofile-statefulset-1
```

You must see those lines among other log reports:

```
SSL certificate created in 58.745 seconds.
The defaultServer server is ready to run a smarter planet..
```

Now, add a value to the map and then get the value:

```
$ > curl "http://localhost:31000/application/map/put?key=key1&value=hazelcast"
$ > while true; do curl localhost:31000/application/map/get?key=key1; echo; sleep 2; done

hazelcast from hazelcast-microprofile-statefulset-0
hazelcast from hazelcast-microprofile-statefulset-0
null from hazelcast-microprofile-statefulset-1
null from hazelcast-microprofile-statefulset-1
```

As can be seen, the data is inserted by `hazelcast-microprofile-statefulset-0` and not shared with the other node. Here is where Hazelcast comes into action.

Before going in to the next step, kill active pods under `initial` directory by running:

```
$ > kubectl delete –f kubernetes.yaml
```

Hazelcast Caching Among Kubernetes Pods

Now we will use Hazelcast caching among the pods. Update the pom.xml file by adding those dependencies:

```
<dependency>
             <groupId>com.hazelcast</groupId>
             <artifactId>hazelcast</artifactId>
             <version>${version.hazelcast}</version>
</dependency>
<dependency>
             <groupId>com.hazelcast</groupId>
             <artifactId>hazelcast-kubernetes</artifactId>
             <version>${version.hazelcast-kubernetes}</version>
</dependency>
```

Then modify `MapApplication.java` and create an application scoped Hazelcast instance inside the class:

```
…

…

import com.hazelcast.config.Config;

import com.hazelcast.config.JoinConfig;

import com.hazelcast.core.Hazelcast;

import com.hazelcast.core.HazelcastInstance;

import javax.enterprise.inject.Produces;

public class MapApplication extends Application {
     @Produces
     HazelcastInstance create() {
          Config config = new Config();
          config.getGroupConfig().setName("MP-GUIDE");
          JoinConfig joinConfig = config.getNetworkConfig().getJoin();
          joinConfig.getMulticastConfig().setEnabled(false);
          joinConfig.getKubernetesConfig().setEnabled(true);
          return Hazelcast.newHazelcastInstance(config);
     }
}
```

Then modify `MapManager.java` such that map is fetched from the Hazelcast instance:

```
…
…
import javax.inject.Inject;
import com.hazelcast.core.HazelcastInstance;
@ApplicationScoped
public class MapManager {

     @Inject
     HazelcastInstance instance;

     //Map<String,String> keyValueStore = new ConcurrentHashMap<>();

     private Map<String,String> retrieveMap() {
          return instance.getMap("map");
     }
     …
     …
     …
}
```

Before deploying on kubernetes, create `rbac.yaml` file as in the `final` directory. Role Based Access Controller(RBAC) configuration is used to give access to Kubernetes Master API from pods which runs microservices. Hazelcast requires a read access to autodiscover other hazelcast members and form hazelcast cluster.

rbac.yaml:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: default-cluster
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: view
subjects:
  - kind: ServiceAccount
    name: default
    namespace: default
```

Rebuild the app and create new image:

```
$ > mvn clean package -P docker-image
```

Run the following commands to deploy the resources as defined in kubernetes.yaml and rbac.yaml in the specified order:

```
$ > kubectl apply -f rbac.yaml
```

```
$ > kubectl apply -f kubernetes.yaml
```

Run the following command to check the status of your pods:

```
$ > kubectl get pods
```

Even if the status of the nodes are `Running`, they might be not started yet. Check the pod logs to be sure they're ready:

```
$ > kubectl logs hazelcast-microprofile-statefulset-0

$ > kubectl logs hazelcast-microprofile-statefulset-1
```

You must see those lines among other log reports. If not, wait for servers to be started and try again:

```
SSL certificate created in 48.248 seconds.
The defaultServer server is ready to run a smarter planet..
```

Now we expect all nodes to give the same value for the same key put on the map by a particular pod. Let's try:

```
$ > curl "http://localhost:31000/application/map/put?key=key_1&value=hazelcast"
$ > while true; do curl localhost:31000/application/map/get?key=key_1;echo; sleep 2; done

hazelcast from hazelcast-microprofile-statefulset-1
hazelcast from hazelcast-microprofile-statefulset-0
hazelcast from hazelcast-microprofile-statefulset-0
hazelcast from hazelcast-microprofile-statefulset-1
```

As can be seen, both nodes give the same value for the key now.

Scaling with Hazelcast

Scale the cluster with one more pod and see that you still retrieve the shared data.

```
$ > kubectl scale statefulset hazelcast-microprofile-statefulset --replicas=3
```

Run following command to see the latest status of the pods

```
$ > kubectl get pods

NAME                                  READY   STATUS        RESTARTS      AGE
hazelcast-microprofile-statefulset-0    1/1    Running       0             9m
hazelcast-microprofile-statefulset-1    1/1    Running       0             9m
hazelcast-microprofile-statefulset-2    1/1    Running       0             6s
hazelcast from hazelcast-microprofile-statefulset-0
hazelcast from hazelcast-microprofile-statefulset-2
```

As you can see, a new pod `hazelcast-microprofile-statefulset-2` has joined the cluster.

Wait for new pod to start and then run the following command again to see the output:

```
$ > while true; do curl "http://localhost:31000/application/map/get?key=key_1";

echo; sleep 2; done
```

```
hazelcast from hazelcast-microprofile-statefulset-1
hazelcast from hazelcast-microprofile-statefulset-2
```

As you can see, `hazelcast-microprofile-statefulset-2` is returning correct data.

Testing Microservices That Are Running on Kubernetes

Create a testing class under `/initial/src/test/java/io/openliberty/sample/system` named `MapResourceTest.java`. The contents of the test file are available under the `final` directory.

The test makes sure that the **/put** endpoint is handled by one of the pods and **/get** methods returns the same data from the other Kubernetes pod.

It first puts a key/value pair to the hazelcast-microprofile microservice and keeps podname in the firstpod variable. In the second part, tests submits multiple **/get** requests until to see that podname is different than the pod which initially handled **/put** request.

In order to run integration tests, you must have a running hazelcast-microprofile microservices in minikube environment. As you have gone through all the previous steps, you already have it.

Run test under `initial` directory:

```
$ > mvn -Dtest=MapResourceTest test
```

If the tests pass, you'll see a similar output to the following:

```
[INFO] -------------------------------------------------------
[INFO] TESTS
[INFO] -------------------------------------------------------
[INFO] Running io.openliberty.sample.system.MapResourceTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 8.48 s -
in io.openliberty.sample.system.MapResourceTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
[INFO] Total time:  19.801 s
[INFO] Finished at: 2019-06-26T17:40:50+03:00
[INFO] ------------------------------------------------------------------
```

Tearing Down the Environment

When you no longer need your deployed microservices, you can delete all Kubernetes resources by running the kubectl delete command: You might need to wait up to 30 seconds as stateful sets kills pods one at a time.

```
$ > kubectl delete -f kubernetes.yaml
```

Windows | Mac

Nothing more needs to be done for Docker Desktop.

Linux

Perform the following steps to return your environment to a clean state.

Point the Docker daemon back to your local machine:

```
$ > eval $(minikube docker-env -u)
```

Stop your Minikube cluster:

```
$ > minikube stop
```

Delete your cluster:

```
$ > minikube delete
```

Conclusion

You have just run a MicroProfile application and created its Docker image. First you ran the app on a container and then deployed it to Kubernetes. You then added Hazelcast caching to the hazelcast-microprofile, and tested with a simple curl command. You also scaled out the microservices and saw that data is shared between microservices. As a last step, you ran integration tests against hazelcast-microprofile that was deployed in a Kubernetes cluster.

350 Cambridge Ave, Suite 100, Palo Alto, CA 94306 USA
Email: sales@hazelcast.com Phone: +1 (650) 521-5453
Visit us at www.hazelcast.com